EXHIBIT 2

Case 1:24-cv-00217-AJT-WEF Document 303-2 Filed 03/14/25 Page 2 of 43 PageID# 7136

Developing a Keyboard Interface

Introduction

Unlike native HTML form elements, browsers do not provide keyboard support for graphical user interface (GUI) components that are made accessible with ARIA; authors have to provide the keyboard support in their code. This section describes the principles and methods for making the functionality of a web page that includes ARIA widgets, such as menus and grids, as well as interactive components, such as toolbars and dialogs, operable with a keyboard. Along with the basics of focus management, this section offers guidance toward the objective of providing experiences to people who rely on a keyboard that are as efficient and enjoyable as the experiences available to others.

This section covers:

1. Understanding fundamental principles of focus movement conventions used in ARIA design patterns.
2. Maintaining visible focus, predictable focus movement, and distinguishing between keyboard focus and the selected state.
3. Managing movement of keyboard focus between components, e.g., how the focus moves when the `Tab` and `Shift` + `Tab` keys are pressed.
4. Managing movement of keyboard focus inside components that contain multiple focusable elements, e.g., two different methods for programmatically exposing focus inside widgets like radio groups, menus, listboxes, trees, and grids.
5. Determining when to make disabled interactive elements focusable.
6. Assigning and revealing keyboard shortcuts, including guidance on how to avoid problematic conflicts with keyboard commands of assistive technologies, browsers, and operating systems.

Fundamental Keyboard Navigation Conventions

ARIA roles, states, and properties model accessibility behaviors and features shared among GUI components of popular desktop GUIs, including Microsoft Windows, macOS, and GNOME. Similarly, ARIA design patterns borrow user expectations and keyboard conventions from those platforms, consistently incorporating common

conventions with the aim of facilitating easy learning and efficient operation of keyboard interfaces across the web.

For a web page to be accessible, all interactive elements must be operable via the keyboard. In addition, consistent application of the common GUI keyboard interface conventions described in the APG patterns is important, especially for assistive technology users. Consider, for example, a screen reader user operating a tree. Just as familiar visual styling helps users discover how to expand a tree branch with a mouse, ARIA attributes give the tree the sound and feel of a tree in a desktop application. So, screen reader users will commonly expect that pressing the right arrow key will expand a collapsed node. Because the screen reader knows the element is a tree, it also has the ability to instruct a novice user how to operate it. Similarly, voice recognition software can implement commands for expanding and collapsing branches because it recognizes the element as a tree and can execute appropriate keyboard commands. All this is only possible if the tree implements the GUI keyboard conventions as described in the Tree View Pattern.

A primary keyboard navigation convention common across all platforms is that the `tab` and `shift` + `tab` keys move focus from one UI component to another while other keys, primarily the arrow keys, move focus inside of components that include multiple focusable elements. The path that the focus follows when pressing the `tab` key is known as the tab sequence or tab ring.

Common examples of UI components that contain multiple focusable elements are radio groups, tablists, menus, and grids. A radio group, for example, contains multiple radio buttons, each of which is focusable. However, only one of the radio buttons is included in the tab sequence. After pressing the `Tab` key moves focus to a radio button in the group, pressing arrow keys moves focus among the radio buttons in the group, and pressing the `Tab` key moves focus out of the radio group to the next element in the tab sequence.

The ARIA specification refers to a discrete UI component that contains multiple focusable elements as a composite (https://w3c.github.io/aria/#composite) widget. The process of controlling focus movement inside a composite is called managing focus. Following are some ARIA design patterns with example implementations that demonstrate focus management:

- Combobox
- Grid
- Listbox
- Menu and Menubar
- Radio Group
- Tabs
- Toolbar
- Treegrid
- Tree View

Discernible and Predictable Keyboard Focus

Work to complete this section is tracked by issue 217. (https://github.com/w3c/aria-practices/issues/217)

When operating with a keyboard, two essentials of a good experience are the abilities to easily discern the location of the keyboard focus and to discover where focus landed after a navigation key has been pressed. The following factors affect to what extent a web page affords users these capabilities.

1. Visibility of the focus indicator: Users need to be able to easily distinguish the keyboard focus indicator from other features of the visual design. Just as a mouse user may move the mouse to help find the mouse pointer, a keyboard user may press a navigation key to watch for movement. If visual changes in response to focus movement are subtle, many users will lose track of focus and be unable to operate. Authors are advised to rely on the default focus indicators provided by browsers. If overriding the default, consider:
    - something about ... Colors and gradients can disappear in high contrast modes.
    - Users need to be able to easily distinguish between focus and selection as described in Focus VS Selection and the Perception of Dual Focus(↘ #kbd_focus_vs_selection), especially when a component that contains

selected elements does not contain the focus.

- ... other considerations to be added ...

2. Persistence of focus: It is essential that there is always a component within the user interface that is active (document.activeElement is not null or is not the body element) and that the active element has a visual focus indicator. Authors need to manage events that effect the currently active element so focus remains visible and moves logically. For example, if the user closes a dialog or performs a destructive operation like deleting an item from a list, the active element may be hidden or removed from the DOM. If such events are not managed to set focus on the button that triggered the dialog or on the list item following the deleted item, browsers move focus to the body element, effectively causing a loss of focus within the user interface.
3. Predictability of movement: Usability of a keyboard interface is heavily influenced by how readily users can guess where focus will land after a navigation key is pressed. Some possible approaches to optimizing predictability include:
    - Move focus in a pattern that matches the reading order of the page's language. In left to right languages, for example, create a tab sequence that moves focus left to right and then top to bottom.
    - Incorporate all elements of a section of the page in the tab sequence before moving focus to another section. For instance, in a page with multiple columns that has content in a left side bar, center region, and right side bar, build a tab sequence that covers all elements in the left sidebar before focus moves to the first focusable element in the center column.
    - When the distance between two consecutive elements in the tab sequence is significant, avoid movement that would be perceived as backward. For example, on a page with a left to right language, a jump from the last element in the bottom right of the main content to the top element in a left-hand sidebar is likely to be less predictable and more difficult to follow, especially for users with a narrow field of view.
    - Follow consistent patterns across a site. The keyboard experience is more predictable when similar pages have similar focus movement patterns.
    - Do not set initial focus when the page loads except in cases where:
        - The page offers a single, primary function that nearly all users employ

immediately after page load.

- Any given user is likely to use the page often.

Focus VS Selection and the Perception of Dual Focus

Occasionally, it may appear as if two elements on the page have focus at the same time. For example, in a multi-select list box, when an option is selected it may be greyed. Yet, the focus indicator can still be moved to other options, which may also be selected. Similarly, when a user activates a tab in a tablist, the selected state is set on the tab and its visual appearance changes. However, the user can still navigate, moving the focus indicator elsewhere on the page while the tab retains its selected appearance and state.

Focus and selection are quite different. From the keyboard user's perspective, focus is a pointer, like a mouse pointer; it tracks the path of navigation. There is only one point of focus at any time and all operations take place at the point of focus. On the other hand, selection is an operation that can be performed in some widgets, such as list boxes, trees, and tablists. If a widget supports only single selection, then only one item can be selected and very often the selected state will simply follow the focus when focus is moved inside of the widget. That is, in some widgets, moving focus may also perform the select operation. However, if the widget supports multiple selection, then more than one item can be in a selected state, and keys for moving focus do not perform selection. Some multi-select widgets do support key commands that both move focus and change selection, but those keys are different from the normal navigation keys. Finally, when focus leaves a widget that includes a selected element, the selected state persists.

From the developer's perspective, the difference is simple -- the focused element is the active element (document.activeElement). Selected elements are elements that have `aria-selected="true"`.

With respect to focus and the selected state, the most important considerations for designers and developers are:

- The visual focus indicator must always be visible.
- The selected state must be visually distinct from the focus indicator.

Deciding When to Make Selection Automatically Follow Focus

In composite widgets where only one element may be selected, such as a tablist or single-select listbox, moving the focus may also cause the focused element to become the selected element. This is called having selection follow focus. Having selection follow focus is often beneficial to users, but in some circumstances, it is extremely detrimental to accessibility.

For example, in a tablist, the selected state is used to indicate which panel is displayed. So, when selection follows focus in a tablist, moving focus from one tab to another automatically changes which panel is displayed. If the content of panels is present in the DOM, then displaying a new panel is nearly instantaneous. A keyboard user who wishes to display the fourth of six tabs can do so with 3 quick presses of the right arrow. And, a screen reader user who perceives the labels on tabs by navigating through them may efficiently read through the complete list without any latency.

However, if displaying a new panel causes a network request and possibly a page refresh, the effect of having selection automatically focus can be devastating to the experience for keyboard and screen reader users. In this case, displaying the fourth tab or reading through the list becomes a tedious and time-consuming task as the user experiences significant latency with each movement of focus. Further, if displaying a new tab refreshes the page, then the user not only has to wait for the new page to load but also return focus to the tab list.

When selection does not follow focus, the user changes which element is selected by pressing the `Enter` or `Space` key.

Keyboard Navigation Between Components (The Tab Sequence)

As explained in section Fundamental Keyboard Navigation Conventions(⇘ #kbd_generalnav), all interactive UI components need to be reachable via the keyboard. This is best achieved by either including them in the tab sequence or by making them accessible from a component that is in the tab sequence, e.g., as part of a composite component. This section addresses building and managing the tab sequence, and subsequent sections cover making focusable elements that are contained within components keyboard accessible.

The HTML tabindex (https://html.spec.whatwg.org/multipage/interaction.html#the-tabindex-attribute) and SVG2 tabindex (https://www.w3.org/TR/SVG2/struct.html#tabindexattribute) attributes can be used to add and remove elements from the tab sequence. The value of tabindex can also influence the order of the tab sequence, although authors are strongly advised not to use tabindex for that purpose.

In HTML, the default tab sequence of a web page includes only links and HTML form elements, except In macOS, where it includes only form elements. macOS system preferences include a keyboard setting that enables the tab key to move focus to all focusable elements.

The default order of elements in the tab sequence is the order of elements in the DOM. The DOM order also determines screen reader reading order. It is important to keep the keyboard tab sequence and the screen reader reading order aligned, logical, and predictable as described in Discernible and Predictable Keyboard Focus(⇘ #kbd_focus_discernable_predictable). The most robust method of manipulating the order of the tab sequence while also maintaining alignment with the reading order that is currently available in all browsers is rearranging elements in the DOM.

The values of the tabindex attribute have the following effects.

tabindex is not present or does not have a valid value

The element has its default focus behavior. In HTML, only form controls and anchors with an HREF attribute are included in the tab sequence.

tabindex="0"

The element is included in the tab sequence based on its position in the DOM.

tabindex="-1"

The element is not included in the tab sequence but is focusable with element.focus().

tabindex="X" where X is an integer in the range 1 <= X <= 32767

Authors are strongly advised NOT to use these values. The element is placed in the tab sequence based on the value of tabindex. Elements with a tabindex value of 0 and elements that are focusable by default will be in the sequence after elements with a tabindex value of 1 or greater.

Keyboard Navigation Inside Components

As described in section Fundamental Keyboard Navigation Conventions(↘ #kbd_generalnav), the tab sequence should include only one focusable element of a composite UI component. Once a composite contains focus, keys other than `Tab` and `Shift` + `Tab` enable the user to move focus among its focusable elements. Authors are free to choose which keys move focus inside of a composite, but they are strongly advised to use the same key bindings as similar components in common GUI operating systems as demonstrated in APG Patterns.

The convention for where focus lands in a composite when it receives focus as a result of a `Tab` key event depends on the type of composite. It is typically one of the following.

- The element that had focus the last time the composite contained focus. Or, if the composite has not yet contained the focus, the first element. Widgets that usually employ this pattern include grid and tree grid.
- The selected element. Or, if there is no selected element, the first element. Widgets where this pattern is commonly implemented include radio groups, tabs, list boxes, and trees. Note: For radio groups, this pattern is referring to the checked radio button; the selected state is not supported for radio buttons.
- The first element. Components that typically follow this pattern include menubars and toolbars.

The following sections explain two strategies for managing focus inside composite elements: creating a roving tabindex and using the aria-activedescendant property.

Managing Focus Within Components Using a Roving tabindex

When using roving tabindex to manage focus in a composite UI component, the element that is to be included in the tab sequence has tabindex of "0" and all other focusable elements contained in the composite have tabindex of "-1". The algorithm for the roving tabindex strategy is as follows.

- When the component container is loaded or created, set `tabindex="0"` on the element that will initially be included in the tab sequence and set `tabindex="-1"` on all other focusable elements it contains.
- When the component contains focus and the user presses a navigation key that

Page Contents

If a component container has an ARIA role that supports the aria-activedescendant (https://w3c.github.io/aria/#aria-activedescendant) property, it is not necessary to manipulate the tabindex attribute and move DOM focus among focusable elements within the container. Instead, only the container element needs to be included in the tab sequence. When the container has DOM focus, the value of aria-activedescendant on the container tells assistive technologies which element is active within the widget.

Assistive technologies will consider the element referred to as active to be the focused element even though DOM focus is on the element that has the aria-activedescendant property. And, when the value of aria-activedescendant is changed, assistive technologies will receive focus change events equivalent to those received when DOM focus actually moves.

The steps for using the aria-activedescendant method of managing focus are as follows.

- When the container element that has a role that supports aria-activedescendant is loaded or created, ensure that:
  - The container element is included in the tab sequence as described in Keyboard Navigation Between Components(⇘ #kbd_general_between) or is a focusable element of a composite that implements a roving tabindex(⇘ #kbd_roving_tabindex).
  - It has `aria-activedescendant="IDREF"` where IDREF is the ID of the element within the container that should be identified as active when the widget receives focus. The referenced element needs to meet the DOM relationship requirements described below.
- When the container element receives DOM focus, draw a visual focus indicator on the active element and ensure the active element is scrolled into view.
- When the composite widget contains focus and the user presses a navigation key that moves focus within the widget, such as an arrow key:
  - Change the value of aria-activedescendant on the container to refer to the element that should be reported to assistive technologies as active.
  - Move the visual focus indicator and, if necessary, scrolled the active element into view.
- If the design calls for a specific element to be focused the next time a user moves focus into the composite with `Tab` or `Shift` + `Tab`, check if aria-activedescendant is referring to that target element when the container loses focus. If it is not, set aria-activedescendant to refer to the target element.

The specification for aria-activedescendant (https://w3c.github.io/aria/#aria-activedescendant) places important restrictions on the DOM relationship between the

focused element that has the aria-activedescendant attribute and the element referenced as active by the value of the attribute. One of the following three conditions must be met.

1. The element referenced as active is a DOM descendant of the focused referencing element.
2. The focused referencing element has a value specified for the aria-owns (https://w3c.github.io/aria/#aria-owns) property that includes the ID of the element referenced as active.
3. The focused referencing element has role of combobox (https://w3c.github.io/aria/#combobox), textbox (https://w3c.github.io/aria/#textbox), or searchbox (https://w3c.github.io/aria/#searchbox) and has aria-controls (https://w3c.github.io/aria/#aria-controls) property referring to an element with a role that supports aria-activedescendant and either:
    1. The element referenced as active is a descendant of the controlled element.
    2. The controlled element has a value specified for the aria-owns (https://w3c.github.io/aria/#aria-owns) property that includes the ID of the element referenced as active.

Focusability of disabled controls

By default, disabled HTML input elements are removed from the tab sequence. In most contexts, the normal expectation is that disabled interactive elements are not focusable. However, there are some contexts where it is common for disabled elements to be focusable, especially inside of composite widgets. For example, as demonstrated in the menu and menubar pattern, disabled items are focusable when navigating through a menu with the arrow keys.

Removing focusability from disabled elements can offer users both advantages and disadvantages. Allowing keyboard users to skip disabled elements usually reduces the number of key presses required to complete a task. However, preventing focus from moving to disabled elements can hide their presence from screen reader users who "see" by moving the focus.

Authors are encouraged to adopt a consistent set of conventions for the focusability of disabled elements. The examples in this guide adopt the following conventions, which both reflect common practice and attempt to balance competing concerns.

1. For elements that are in the tab sequence when enabled, remove them from the tab sequence when disabled.
2. For the following composite widget elements, keep them focusable when disabled:
   - Options in a Listbox
   - Menu items in a Menu or menu bar
   - Tab elements in a set of Tabs
   - Tree items in a Tree View
3. For elements contained in a toolbar, make them focusable if discoverability is a concern. Here are two examples to aid with this judgment.
   1. A toolbar with buttons for moving, removing, and adding items in a list includes buttons for "Up", "Down", "Add", and "Remove". The "Up" button is disabled and its focusability is removed when the first item in the list is selected. Given the presence of the "Down" button, discoverability of the "Up" button is not a concern.
   2. A toolbar in an editor contains a set of special smart paste functions that are disabled when the clipboard is empty or when the function is not applicable to the current content of the clipboard. It could be helpful to keep the disabled buttons focusable if the ability to discover their functionality is primarily via their presence on the toolbar.

One design technique for mitigating the impact of including disabled elements in the path of keyboard focus is employing appropriate keyboard shortcuts as described in Keyboard Shortcuts(↘ #kbd_shortcuts).

Key Assignment Conventions for Common Functions

The following key assignments can be used in any context where their conventionally

associated functions are appropriate. While the assignments associated with Windows and Linux platforms can be implemented and used in browsers running in macOS, replacing them with macOS assignments in browsers running on a macOS device can make the keyboard interface more discoverable and intuitive for those users. In some cases, it may also help avoid system or browser keyboard conflicts.

Function	Windows/Linux Key	macOS Key
open context menu	`Shift` + `F10`	
Copy to clipboard	`Control` + `C`	`Command` + `C`
Paste from clipboard	`Control` + `V`	`Command` + `V`
Cut to clipboard	`Control` + `X`	`Command` + `X`
undo last action	`Control` + `Z`	`Command` + `Z`
Redo action	`Control` + `Y`	`Command` + `Shift` + `Z`

Keyboard Shortcuts

When effectively designed, keyboard shortcuts that focus an element, activate a widget, or both can dramatically enhance usability of frequently used features of a page or site. This section addresses some of the keyboard shortcut design and implementation factors that most impact their effectiveness, including:

1. Understanding how keyboard shortcuts augment a keyboard interface and whether to make a particular shortcut move focus, perform a function, or both.
2. Making key assignments and avoiding assignment conflicts with assistive technologies, browsers, and operating systems.
3. Exposing and documenting key assignments.

Designing the Scope and Behavior of Keyboard Shortcuts

This section explains the following factors when determining which elements and features to assign keyboard shortcuts and what behavior to give each shortcut:

1. Ensuring discovery through navigation; keyboard shortcuts enhance, not replace, standard keyboard access.
2. Effectively choosing from among the following behaviors:
    1. Navigation: Moving focus to an element.
    2. Activation: Performing an operation associated with an element that does not have focus and might not be visible.
    3. Navigation and activation: Both moving focus to an element and activating it.
3. Balancing efficiency and cognitive load: lack of a shortcut can reduce efficiency while too many shortcuts can increase cognitive load and clutter the experience.

Ensure Basic Access Via Navigation

Before assigning keyboard shortcuts, it is essential to ensure the features and functions to which shortcuts may be assigned are keyboard accessible without a keyboard shortcut. In other words, all elements that could be targets for keyboard shortcuts need to be focusable via the keyboard using the methods described in:

- Keyboard Navigation Between Components(⇘ #kbd_general_between)
- Keyboard Navigation Inside Components(⇘ #kbd_general_within)

Do not use keyboard shortcuts as a substitute for access via navigation. This is essential to full keyboard access because:

1. The primary means of making functions and their shortcuts discoverable is by making the target elements focusable and revealing key assignments on the element itself.
2. If people who rely on the keyboard have to read documentation to learn which keys are required to use an interface, the interface may technically meet some accessibility standards but in practice is only accessible to the small subset of them who have the knowledge that such documentation exists, have the extra

time available, and the ability to retain the necessary information.

3. Not all devices that depend on keyboard interfaces can support keyboard shortcuts.

Choose Appropriate Shortcut Behavior

The following conventions may help identify the most advantageous behavior for a keyboard shortcut.

- Move focus when the primary objective is to make navigation more efficient, e.g., reduce the number of times the user must press `Tab` or the arrow keys. This behavior is commonly expected when assigning a shortcut to a text box, toolbar, or composite, such as a listbox, tree, grid, or menubar. This behavior is also useful for moving focus to a section of a page, such as the main content or a complementary landmark section.
- Activate an element without moving focus when the target context of the function is the context that contains the focus. This behavior is most common for command buttons and for functions associated with elements that are not visible, such as a "Save" option that is accessible via a menu. For example, if the focus is on an option in a listbox and a toolbar contains buttons for moving and removing options, it is most beneficial to keep focus in the listbox when the user presses a key shortcut for one of the buttons in the toolbar. This behavior can be particularly important for screen reader users because it provides confirmation of the action performed and makes performing multiple commands more efficient. For instance, when a screen reader user presses the shortcut for the "Up" button, the user will be able to hear the new position of the option in the list since it still has the focus. Similarly, when the user presses the shortcut for deleting an option, the user can hear the next option in the list and immediately decide whether to press the delete shortcut again.
- Move focus and activate when the target of the shortcut has a single function and the context of that function is the same as the target. This behavior is typical when a shortcut is assigned to a button that opens a menu or dialog, to a checkbox, or to a navigation link or button.

Choose Where to Add Shortcuts

Work to draft content for this section is tracked in issue 219. (https://github.com/w3c/aria-practices/issues/219)

The first goal when designing a keyboard interface is simple, efficient, and intuitive operation with only basic keyboard navigation support. If basic operation of a keyboard interface is inefficient, attempting to compensate for fundamental design issues, such as suboptimal layout or command structure, by implementing keyboard shortcuts will not likely reduce user frustration. The practical implication of this is that, in most well-designed user interfaces, the percentage of functionality that needs to be accessible via a keyboard shortcut in order to create optimal usability is not very high. In many simple user interfaces, keyboard shortcuts can be entirely superfluous. And, in user interfaces with too many keyboard shortcuts, the excess shortcuts create cognitive load that make the most useful ones more difficult to remember.

Consider the following when deciding where to assign keyboard shortcuts:

1. To be written.

Assigning Keyboard Shortcuts

When choosing the keys to assign to a shortcut, there are many factors to consider.

- Making the shortcut easy to learn and remember by using a mnemonic (e.g., `Control` + `S` for "Save") or following a logical or spacial pattern.
- Localizing the interface, including for differences in which keys are available and how they behave and for language considerations that could impact mnemonics.
- Avoiding and managing conflicts with key assignments used by an assistive technology, the browser, or the operating system.

Methods for designing a key shortcut scheme that supports learning and memory is beyond the scope of this guide. Unless the key shortcut scheme is extensive, it is likely sufficient to mimic concepts that are familiar from common desktop software, such as browsers. Similarly, while localization is important, describing how to address it is left to other resources that specialize in that topic.

The remainder of this section provides guidance balancing requirements and concerns

related to key assignment conflicts. It is typically ideal if key assignments do not conflict with keys that are assigned to functions in the user's operating system, browser, or assistive technology. Conflicts can block efficient access to functions that are essential to the user, and a perfect storm of conflicts can trap a user. At the same time, there are some circumstances where intentional conflicts are useful. And, given the vast array of operating system, browser, and assistive technology keys, it is almost impossible to be certain conflicts do not exist. So it is also important to employ strategies that mitigate the impact of conflicts whether they are intentional or unknown.

> **Note**
>
> In the following sections, `meta` key refers to the `Windows` key on Windows-compatible keyboards and the `Command` key on MacOS-compatible keyboards.

Operating System Key Conflicts

It is essential to avoid conflicts with keys that perform system level functions, such as application and window management and display and sound control. In general, this can be achieved by refraining from the following types of assignments.

1. Any modifier keys + any of `Tab`, `Enter`, `Space`, or `Escape`.
2. `Meta` key + any other single key (there are exceptions, but they can be risky as these keys can change across versions of operating systems).
3. `Alt` + a function key.

In addition, there are some important application level features that most applications, including browsers, generally support. These include:

1. Zoom
2. Copy/Paste
3. ... to be continued ...

Assistive Technology Key Conflicts

Even though assistive technologies have collectively taken thousands of key assignments, avoiding conflicts is relatively easy. This is because assistive technologies have had to develop key assignment schemes that avoid conflicts with both operating systems and applications. They do this by hijacking specific keys as modifiers that uniquely define their key commands. For example, many assistive technologies use the `Caps Lock` key as a modifier.

Deflect assistive technology key conflicts by steering clear of the following types of assignments.

1. `Caps Lock` + any other combination of keys.
2. `Insert` + any combination of other keys.
3. `Scroll Lock` + any combination of other keys.
4. macOS only: `Control` + `Option` + any combination of other keys.

Browser Key Conflicts

While there is considerable similarity among browser keyboard schemes, the patterns within the schemes are less homogenous. Consequently, it is more difficult to avoid conflicts with browser key assignments. While the impact of conflicts is sometimes mitigated by the availability of two paths to nearly every function -- keyboard accessible menus and keyboard shortcuts, avoiding conflicts with shortcuts to heavily used functions is nonetheless important. Pay special attention to avoiding conflicts with shortcuts to:

1. Address or location bar
2. Notification bar
3. Page refresh
4. Bookmark and history functions
5. Find functions

Intentional Key Conflicts

While avoiding key conflicts is usually desirable, there are circumstances where

intentionally conflicting with a browser function is acceptable or even desirable. This can occur when the following combination of conditions arises:

- A web application has a frequently used function that is similar to a browser function.
- Users will often want to execute the web application function.
- Users will rarely execute the browser function.
- There is an efficient, alternative path to the browser function.

For example, consider a save function that is available when the focus is in an editor. Most browsers use … to be continued …

W3C Web Accessibility Initiative (WAI)
Strategies, standards, and supporting resources to make the Web accessible to people with disabilities.

Copyright © 2024 World Wide Web Consortium (W3C®). See Permission to Use WAI Material.

ACCESSIBILITY DEVELOPER GUIDE

How to implement websites that are ready for keyboard only usage

Fortunately, to make sure a website is ready to be used by keyboard only, there is very little you need to consider. Still, a lot can go wrong if not paying attention, and bad old habits of some web developers seem to be hard to overcome.

On this page

Standard functionality

As HTML is fully keyboard only ready, all standard HTML controls offer built-in support for keyboard interaction. This is provided by the browser. So if you stick to using proper semantic HTML markup, you're all fine.

Thus, if you haven't done this yet, go back and read Semantics and their importance for accessibility.

Custom functionality

If you need to provide custom functionality on your website (for example a fancy JavaScript widget that does not have an HTML equivalent), you may need to optimise it specifically for keyboard interaction.

Focusability

First, be sure any relevant element is focusable by keyboard:

- Either by using an HTML element that inherently is focusable (for example a link or a button).
  - Whenever an element should be interacted with, this is the proper way to go.
- Or by adding the `tabindex="0"` attribute, for example `<div tabindex="0">`
  - This should only be used in situations when an element shall be focusable, but not interactive (a very rare case).
- A special case are `<a>` tags: if they have an `href` attribute, they are focusable; otherwise, in most browsers they are not.

Keyboard operability

Then, implement any functionality as needed by adding appropriate events, for example `onclick` or similar:

- Be sure to use browser-independent events.
  - An `onclick` event is triggered by all sorts of devices (regardless whether it's fired upon a mouse click or by pressing a keyboard's `Enter` key).
  - On the other side, a `hover` event can only be triggered by some devices (for example mouse) and not by others (for example keyboard)
    - A well-known accessibility problem is dropdown menus opening on `hover` only, so no keyboard user will ever be able to open it.
- Avoid listening to specific keyboard keys, unless you really want to provide device-specific functionality.
  - For example, listening to a `keypress` event and looking for a key code `13` (`Enter` key) will only catch a real keyboard's `Enter` key.
    - But it won't catch a screen reader's `Enter` key, as pressing `Enter` with a screen reader running will typically fire a `click` event instead.

  - As described above, it's usually better to listen to a `click` event here.

- Be aware that many keys are only available on full-fledged keyboards of desktop computers.
  - Be sure not to bind functionality to such specific keys (for example modifier keys like `Alt`), or provide alternative ways to use the functionality.
  - Even arrow keys aren't available on many devices, for example touch screen smartphones.
  - Especially combinations of keys can be really hard (if not impossible) to be pressed by motor impaired people.

Standards over custom

In our opinion, there are very few cases which really need optimisation for keyboard interaction. Always think twice and look for a standard behaviour first that may fit your need (even if it looks very different on first sight).

Focus management

Another facet of keyboard optimisation is setting the focus properly if needed.

This can be explained especially well by looking at the dialog pattern. Indeed, our guide has such a dialog implementaion available as a fully working example, so if you are really curious and want to learn more about this, skip ahead and read Dialog widget (or: modal, popup, lightbox, alert).

Setting focus to an element

For example, when a user clicks on a button "Show terms and conditions (dialog)", then it is important that the keyboard focus is set into the dialog properly. Otherwise, the keyboard focus usually is hidden somewhere "behind" the dialog. If users then want to interact with the dialog (be it filling out some displayed form elements, clicking a link within its content, or simply pressing its close button), they are forced to press `Tab` again and again until (hopefully) the focus suddenly appears in the dialog.

Specifically, the first content element of the dialog should be focused, due to the following reason: users usually make their way from top to bottom of a page, be it while reading content or moving the keyboard focus. Placing the focus on the first content element does not interrupt this workflow. It usually is a good practice to define the dialog's close button to be this first element:

```
$("button#show-dialog").click(function() {
  $("div#dialog button.close").focus();
});


<button id="show-dialog">
  Show terms and conditions (dialog)
</button>

<div id="dialog">
  <button class="close">Close dialog</button>

  <p>Here is some content...</p>
</div>
```

Avoiding focus on background elements

Dialogs typically are placed on top of the page's other content, often darkening the background. As explained above, keeping the focus placed in the background after opening a dialog is nasty for keyboard users. So we should take care that this does not happen by accident again when the user is interacting with the dialog.

One way is to trap the focus within the dialog: this means that after the last focusable element of the dialog loses focus, the first element is focused again. This way, if users have interacted with the form and then want to close it again using the close button on top of the dialog, they do not have to tab back to the top (using `Shift + Tab`), but intuitively reach the close button again automatically.

But there is a big disadvantage with this technique: users will not be able at all to leave the page content using `Tab` key, for example if they want to reach the address bar. Sure, there are other ways to do this (for example pressing `F6` or `Ctrl + L`), but some users do not know about this.

A better solution is to not trap the focus, but to offer another close button at the bottom of the dialog. This way, the user won't have the need anymore to press `Tab` again, in hope to reach the close button at the top.

```
<div id="dialog">
  <button class="close">Close dialog</button><!-- Usually an

  <p>Here is some content...</p>

  <button class="confirm">Confirm</button><!-- This confirms
  <button class="cancel">Cancel</button><!-- This cancels the
</div>
```

Resetting focus

As soon as the user closes the dialog, the focus needs to be set back to the element it initially had been before opening the dialog. In our example, this was the "Show terms and conditions (dialog)" button. Otherwise, the focus gets "lost", and it is automatically set by the browser to the very beginning of the page.

In some cases, the initial element may not be available anymore. For example, in a shopping cart, when removing an item opens a dialog, asking "Are you sure you want to delete this item?". If confirmed, the page would do some AJAX, removing the item from the page, then closing the confirmation dialog. In this case, you should place the focus somewhere that is still reasonable to the user. For example, place it on the next item's remove button. Or if you are displaying some notification like "Item successfully removed from shopping cart" or "No items left in shopping cart".

Be careful with setting focus

Only set focus if it clearly is not feasible to leave it where it currently is. So you never want to bug your users by moving focus around nervously!

An example for bad focus management is a tablist like the following: after activating a tab, the focus would be automatically set to the selected tab's content. This may seem useful at the first glance: users selecting a tab typically want to interact with its content directly, do not they? But at the second glance, if visual users simply want to toggle through the available tabs' contents quickly, they really need the focus to stay on the tabs.

However, in our dialog example above, leaving the focus in the darkened (or even invisible) background surely is no option. Thus, in this case, moving the focus into the dialog definitely is the right way to go.

Conclusion

Optimising a website for keyboard only usage needs some awareness of the typical pitfalls. Providing proper HTML elements helps truly a lot already, but the full deal requires some additional developer skills.

All of the code examples in our guide are fully accessible to the most common assistive software and devices and can provide you a lot of inspiration. If you are really curious and want to learn more about this, skip ahead and read Examples of accessibility patterns.

👈 How to browse websites using ... Introduction to screen reader u... 👉

Page history:

An initiative of:

In cooperation with:

Further information:
Privacy policy - License - Feed - GitHub - Facebook - Twitter

Based strictly on W3C's WCAG 2.1 and ARIA | Generously hosted by Netlify

✎ Technique: Keyboard accessible links

Links (<a> elements) are focusable by default, so it should normally be possible to activate them using the keyboard. However, under certain circumstances, they are not focusable: Unless you are deliberately trying to "disable" a link, make sure it does not take the form of any of these examples.

Examples

`href` removal

Sometimes <a> elements are used as if they're <button> elements, removing the `href` attribute because it is redundant. This will make the link unfocusable and inoperable by keyboard. Unless you want that to happen, don't remove the `href`.

```
<a>click me to execute JavaScript event</a>
```

You can use the following CSS in your developer tools to highlight links that do not have

an `href`. The CSS produces a red highlight around links that do not work using a keyboard.

```
a:not([href]) {
  border: 2px solid #c00;
}
```

Inclusion of `tabindex="-1"`

You can control the focusability of any element with the `tabindex` attribute. The value `-1` will make the element unfocusable by the user. However, the element will still be focusable through JavaScript using the `focus()` method.

```
<a href="path/to/url" tabindex="-1">I'm currently unfocusable</a>
```

See also:

Controls | Links | Keyboard | Speech recognition | Switch-based input | Blind | Motor impairment | HTML | JavaScript | Custom widgets and controls ✎ | Support keyboard interaction ✎ | Operable

Contact Us

About Digital Accessibility Services (DAS)

digitalaccessibility@harvard.edu

Admin login

HARVARD UNIVERSITY

Copyright © 2024 The President and Fellows of Harvard College

Accessibility | Digital Accessibility | Report Copyright Infringement | Info Practices

evinced Digital Accessibility Knowledge Base

System Validations
Mobile Validations
Elements
A11y Concepts
Components
Evinced, Inc.
Blog

keyboard accessible

* All of the UI functionality must be available and accessible by keyboard

Who might be affected

Keyboard

Description

When you coerce different semantics on a UI element by using the role attribute or any other way, **it will not give it all the semantic characteristics** including focusability. You must make sure that an element is focusable for it to be keyboard accessible. Read more about semantic elements and roles on A11y concepts > Semantic DOM

Quick Fixes

Use the proper semantic tag instead of "div" or "span".

```
1 ✓ <button class="button">Submit</button>
2 ✓
3 ✓ <a href="/some/path">Page Name</a>
4
5 ✗ <div role="button" class="button">Submit</div>
6 ✗
7 ✗ <div role="link">Page Name</div>
```

If it is inevitable to use a non-semantic element, add an `tabindex` attribute to it with the value "0" to make it accessible to keyboards.

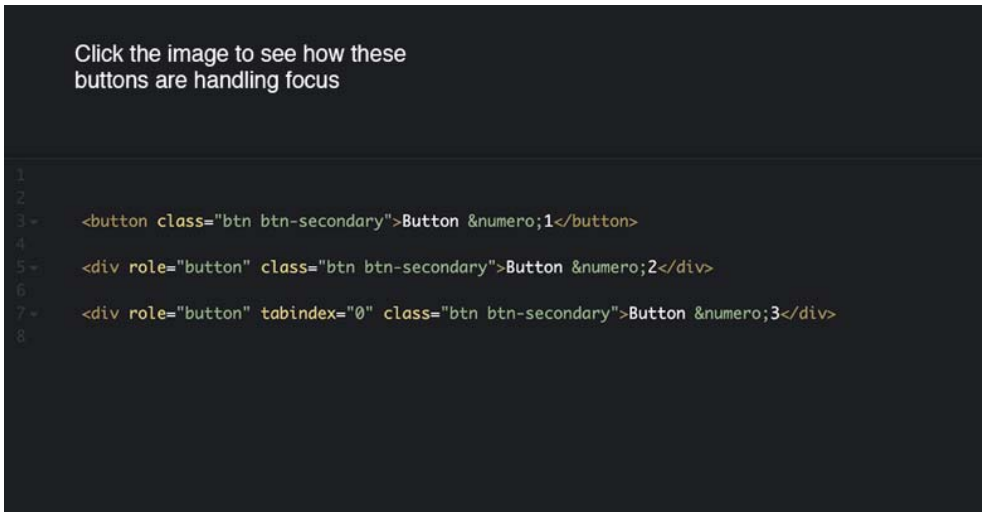
```
1 ✓ <div role="button" class="button" tabindex="0">S
2 ✓
3 ✓ <div role="link" tabindex="0">Page Name</div>
```

Learn more about semantic elements and roles on Ally concepts > Semantic DOM

How Users Are Affected

When a UI element lacks the semantics or attributes needed to be focusable, it is excluded from the keyboard tabbing sequence, and therefore inaccessible for keyboard users.

Play the animation below to see the behavior of non-focusable elements.



WCAG Success criteria

This issue might cause elements to fail one or more of the following Success criteria:
1.3.3 Sensory Characteristics (A) | 2.1.1 Keyboard (A) | 4.1.2 name, role, value (A)

Recommended Reading

- Semantic DOM Evinced KB
- It's Just A Matter Of Semantics By Evinced
- Future Web Accessibility: HTML5 Semantic Elements By WEBAIM
- Accessibility Through Semantic HTML By Laura Kalbag


evinced
Products . Knowledge Base . Blog . About Us .

IOWA Information Technology Services

Accessibility Tips: Images and Non-Text Elements

Images and non-text elements

- All <img> and <area> elements include ALT attributes
- ALT attributes provide content and functionality (i.e., linking, focusability) equivalent to the <img> or <area> element
- Long text descriptions are provided for non-text elements that require detailed explanation
- Decorative images include empty ALT text (alt="") and no TITLE attribute
- Images included using CSS (e.g., background images, list-style-images) do not convey important information or functionality, including navigation or branding
- Embedded and other objects (e.g., multimedia, applets, Flash components, etc.) are accompanied by appropriate text alternatives and references
- Pages with visual CAPTCHAs include CAPTCHAs that use a different modality

Last updated: Wednesday, September 4, 2024

Article number: 4651

IOWA

Information Technology Services
Information Technology Services (ITS)
2800 University Capitol Centre
Iowa City, IA 52242
Walk-in hours and location
319-384-4357
its-helpdesk@uiowa.edu
Help Desk Online Chat
Website Feedback
Privacy Information
© 2024 The University of Iowa

Accessibility

Dimagi's product design team firmly believes that web and software experiences should be accessible for everyone, regardless of abilities or impairments.

Overview

Accessible products give every user the same benefits regardless of ability and can adapt to any user in any context. Disabilities can be situational (an end user with typical vision may struggle to view their screen in a bright or sunny environment), temporary (a person with a broken wrist may not be able to type in HQ but will regain ability when healed), and long-lasting (barriers from birth, an illness, disease, accident, or developed over age; some may not consider themselves to have disabilities even if they experience functional limitations). We design and build products for everyone.

"Everyone" includes

- 18% of adults with vision impairment (source (http://www.who.int/news-room/fact-sheets/detail/blindness-and-visual-impairment))
- 8.6% of adults with hearing trouble (16.8% of U.S.

adults!) (source (https://www.who.int/deafness/en/) and source (https://www.cdc.gov/nchs/fastats/disability.htm))
- 15% of adults with physical functioning difficulty (source (http://www.who.int/news-room/fact-sheets/detail/disability-and-health) and source (https://www.cdc.gov/nchs/fastats/disability.htm))
- 4.4% of adults with cognitive disabilities (source (https://www.cdc.gov/nchs/fastats/disability.htm))

Consider the following barriers when designing and coding

- Blindness (may use screen reader, lack access to mouse, rely on text over visual information
- Low-Vision (may use screen readers/magnifiers, high contrast or monochrome displays; maximize readability and visual clarity; try NoCoffee plugin
- Color-blindness (may not be able to differentiate between some colors; test designs in a color-blind simulator)
- Deaf and Hard-of-Hearing (may rely on captioning and alternative representations of audio
- Physical Disabilities (may rely on keyboards, trackballs, voice recognition or assistive tech, may lack mouse access
- Cognitive Disabilities (may have limited working memory, problem solving, attention, reading/linguistic/visual comprehension, sensitivity to flashing content; try to design linearly and focus on heuristics that have to do with cognitive load and memory)

Color-blindness: A11y Color Contrast Standards

A11y check

AAA Contrast 7.01:1

AAA Contrast 11.07:1

Quick ways to test web accessibility

- **Keyboard Navigation:** Test if modified code is navigable by keyboard, an effective way to ensure page navigability and screen reader visibility. More information and tips here (https://webaim.org/techniques/keyboard/#testing).
- **Explore the Accessibility Tree:** Use Chrome DevTools' accessibility feature to view the accessibility tree of a page under development, understanding how page elements are included, visible, and organized. This is especially helpful for understanding what a specific element is, how it's labeled, and how it's defined in relation to other elements on the screen.
- **Use a Screen Reader:** Free options like VoiceOver for Mac, ORCA for Linux, and NVDA for Windows can assist in testing accessibility. Warning: different screen readers can read the same webpage differently so using one is a good launching point but it is also recommended to utilize different screen readers in final testing.
- **Check color contrast:**We aim for AA compliant color contrast. You can check color contrast in tools such as https://webaim.org/resources/contrastchecker/ (https://webaim.org/resources/contrastchecker/).
- **WAVE Browser Extension:** WAVE (https://wave.webaim.org/) which identifies Web Content Accessibility Guideline (WCAG) errors

- Keep in mind that this list is not exhaustive and covers only a portion of what makes a site accessible to all users. Ensuring accessibility is an ongoing commitment to creating inclusive digital experiences for everyone.

Focusability, Tabbability, and Visibility

tabindex Attribute

The tabindex attribute specifies the tab order of an element and determines whether an element can receive keyboard focus. Here are some scenarios where you might use tabindex:

- Focusable but Not Natively Focusable Elements: Some elements like or are not natively focusable, but you might want them to receive focus for user interaction. In such cases, you can add tabindex="0" to make them focusable. Note: we avoid using custom focus order using tab index with values above 0. Instead, make sure DOM elements are ordered correctly for navigability, then modify visual positioning with CSS if needed.
- Remove Focusability: to explicitly remove an item set tabindex="-1"
- Dynamic Focus Control: If you have dynamically generated or hidden elements that need to become focusable under certain conditions, you can dynamically add or remove the tabindex attribute.

aria-hidden Attribute

The aria-hidden attribute indicates whether an element is visible or hidden to assistive technologies. It does not affect the visual rendering of the page. Here are some scenarios where you might use aria-hidden:

- Hiding Decorative Elements: If you have decorative or presentational elements that should be hidden from screen readers, you can use aria-hidden="true".
- Dynamic Content Updates: When dynamically updating content, you may want to temporarily hide or show certain elements from assistive technologies without altering their visual display.
- Modal Dialogs or Popovers: When displaying modal dialogs or popovers, you may want to hide the underlying content from screen readers to prevent confusion.

When to Use Each

Use tabindex when you need to remove or add tab focusability to an element.

Use aria-hidden when you need to hide elements from assistive technologies while keeping them visible on the page or dynamically toggle their visibility.

Common Mistakes (and how to fix them)

1. **Using Non-Semantic HTML Elements for Interactive Components**
   Error: Using or elements for interactive components like buttons or links instead of using native HTML elements.

```
<!-- Incorrect: Using <div> for button -->

<div onclick="handleClick()">Click me</div>

<!-- Correct: Using <button> for button -->

<button onclick="handleClick()">Click me</butt
```

Fix: Replace non-semantic elements with appropriate semantic HTML elements like <button> for buttons and <a> for links. This fix also ensure that by creating interactive elements that cannot be accessed or operated using the keyboard.

2. **Missing or Incomplete ARIA (Accessible Rich Internet Applications) Attributes**
   Error: Failing to provide necessary ARIA attributes or providing incomplete information, which can result in screen readers not conveying the correct information to users.

```
<!-- Incorrect: Missing aria-label attribute f

<button>×</button>

<!-- Correct: Button with aria-label to convey

<button aria-label="Close">×</button>
```

   Fix: Include essential ARIA attributes like aria-label or aria-labelledby to provide meaningful information to assistive technologies when symbols or other visual information convey information that is otherwise not present.

3. **Inconsistent Use of ARIA Attributes**
   Error: Inconsistently using ARIA attributes across similar UI components, leading to confusion for users relying on assistive technologies.

```
<!-- Incorrect: Inconsistent use of aria-hidde

<div id="modal1" aria-hidden="true">Modal 1 Co

<div id="modal2">Modal 2 Content</div>

<!-- Correct: Consistent use of aria-hidden --

<div id="modal1" aria-hidden="true">Modal 1 Co

<div id="modal2" aria-hidden="true">Modal 2 Co
```

4. **Overusing ARIA attributes**
   Error: Using ARIA attributes unnecessarily can clutter the code and potentially confuse users or interfere with native accessibility features. This is often a crutch needed when not using semantic HTML (Semantic means "relating to meaning". Writing semantic HTML means using HTML elements to structure content based on each element's meaning, not its appearance.)

   ```
   <!-- Incorrect: Using a clickable div rather t

   <div role="button" tabindex="0" aria-label="Cl

   <!-- Correct: Minimal use of ARIA attributes -

   <button>Click me</button>
   ```

   Fix: Use ARIA attributes only when necessary to supplement or enhance the accessibility of UI components, avoiding unnecessary or redundant attributes.

By avoiding these common coding errors and following best practices for ARIA implementation, you can ensure that your front-end code adheres to ARIA guidelines and provides a more accessible user experience for all users.

Case 1:24-cv-00217-AJT-WEF Document 303-2 Filed 03/14/25 Page 43 of 43 PageID# 7177

Quick Links

- Bootstrap 5 Docs (https://getbootstrap.com/docs/5.3/getting-started/)
- Knockout.js Docs (https://knockoutjs.com/documentation/introduction.html)
- Laws of UX (https://lawsofux.com/)

Thanks for reading the style guide! <3